Expressivity of the Microlog Language

Mario Wenzel

Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik,
Von-Seckendorff-Platz 1, D-06099 Halle (Saale), Germany
mario.wenzel@informatik.uni-halle.de

Abstract. In this paper we analyse the expressiveness of our programming language Microlog. Microlog is a variant of Datalog with a strict call-convention to allow for interactive declarative logic-programming with very little "plumbing code". A focus on memory usage, both program memory and dynamic memory, allows Microlog programs to run on microprocessors with as little as 1 KB of RAM and a few KB of program memory, like an Arduino nano.
The analysis of Microlog's expressiveness is conducted in reference to the agent model as defined by Russel and Norvig. Example programs illustrate the problems that might be tackled using the Microlog language.

Keywords: Datalog · Logic Programming · Microlog · Microcontroller · Intelligent Agent.

1 Introduction

We usually prefer declarative logic programming (LP) over imperative programming because the LP languages have mathematically precise semantics based on logic, which makes programs easier to verify and programming arguably easier to teach. Even primary school children can handle deduction-based systems (like Prolog) but struggle with the specifics of backtracking [Enn82], as do students, especially when side-effects are involved. [SW19] Complex interactive software systems, while often written in an imperative language, usually include a core of decision procedures, that govern the interactions with the outside world, which resemble a logic program; sometimes unbeknownst to the programmers, sometimes embedded explicitly as a (third-party) component.

As microcontrollers have become very cheap (Arduino, for example), they have found their way to hobbyists' workshops and school and university courses. When programming for microcontrollers there are usually not enough resources to properly separate concerns, so that we neither find special LP languages for microcontrollers, nor resource-friendly implementations of LP languages embedded into C or C++, which are the de-facto standard microcontroller programming languages. As memory management and debugging on microcontrollers are difficult problems, non-professionals struggle to create complex programs.

Our programming language Microlog aims at hobbyists and educators, that want to describe somewhat complex decision procedures in a logic programming

language without having to deal with the specific challenges of microcontroller programming, like memory management (both in memory for dynamic object creation and stack memory for function call depth), or modelling of abstractions and architecture.

In this paper we want to show that Microlog is expressive enough to define at least some of the classes of intelligent agents as defined by Norvig and Russel. [RN10]

In Section 2 we recapitulate the Microlog language, its semantics and some concrete syntax in order to present and explain the example programs for the different agent types in section 3. In Section 4 we present the different possible encodings of state, as described by Norvig and Russel and relate them to the different compilation techniques presented by us.

2 Microlog Recap

The Microlog language is modelled after the Dedalus$_0$ language [Alv+10]. Both Dedalus$_0$ and Microlog are based on Datalog. A Microlog program is a finite set of rules of the form $A \leftarrow B_1 \wedge \cdots \wedge B_n$, where the head literal A is a positive atomic formula and the body literals B_i each are either a positive atomic formulas $p(t_1, \ldots, t_m)$, a negated atomic formulas $\neg p(t_1, \ldots, t_m)$, or quantifier-free formula in some chosen theory of first-order logic (e. g., arithmetic comparisons $e_l \square e_r$ with the expressions e_l, e_r, and a comparison operator $\square$ over the integers). Rules must be range-restricted, i.e., all variables that appear anywhere in the rule must appear also in a body literal with a positive atomic formula. This ensures that all variables are bound to a value when the rule is applied. Besides variables and constants of the theory, we also allow library constants, like `HIGH` or `LOW` for the Arduino digital input levels, which are prefixed with `#`.

In order to explicitly model time, Microlog programs have the following syntactic restriction:

- Every predicate must have a first argument from the domain of the natural numbers which we refer to as the *timestamp*. This has the meaning that some fact $p(\ldots)$ is true in timestamp n iff $p(n, \ldots)$ is in the minimal model of our program. We refer to the selection of all facts with a certain timestamp as a *state* $\mathcal{S}_n$.
- All normal literals in a rule body must have as their timestamp the same special variable $\mathcal{T}$, as rules may only rely on facts from a single timestamp.
- The rule head either shares the same $\mathcal{T}$ as its timestamp (as in Dedalus, this is called a *deductive rule*)
- or it has the timestamp $\mathcal{T}'$ and the literal succ($\mathcal{T}$,$\mathcal{T}'$) is part of the rule body (as in Dedalus, this is called an *inductive rule*).
- Rules without body are only allowed as initial facts if their timestamp is 0.

A Microlog must interface with libraries for querying input devices and performing actions on output devices as well. For each function f that can be called, there is a special predicate call_f with a reserved prefix "call_ ". The predicate

has arguments of the function to be called, arguments for the return values, and of course the standard time argument. E.g., derived facts about the predicate call_digitalRead($\mathcal{T}'$,*Port*,?) lead to the corresponding calls of the interface function `digitalRead` in the following state $\mathcal{T}'$ with the argument *Port*. In the following state a fact like ret_digitalRead($\mathcal{T}$,*Port*,`#HIGH`) (depending on the concrete return value) is available.

The set-semantics ensures that duplicate calls are eliminated, i.e., even if there are different ways to deduce the fact, only one call is done. For the output positions that are not assigned a value in the "call_ "-predicate the special marker ? to achieve a consistent argument list. One could view this as an existentially quantified anonymous variable with the promise that for each ? in a derived call_-fact, there will be some return value in the corresponding ret_-fact.

This call-convention is a combination of action atoms [Bas+10] with its function calls in the rule head, and the earlier proposal of external atoms [CI05] that allow for pure function calls in the rule body. Special symbols in the rule head as place holders for invented values of a logic program have been used in [HY90] to denote fresh identifiers.

We use some syntactic sugar and to make it easier to work with the syntactic restrictions and borrow some concrete syntax from the Dedalus language:

Unsugared Version	Sugared Version
Deductive Rules: the time argument is left out in the rule head and every sub-goal.	
$p(\mathcal{T},X) \leftarrow q(\mathcal{T},X,Y) \wedge p(\mathcal{T},Y).$	`p(X) :- q(X, Y), p(Y).`
Inductive Rules: the suffix "@next" is added to the rule head and the time argument is left out in the rule head and every sub-goal and we leave out the succ predicate.	
$p(\mathcal{T}',X) \leftarrow q(\mathcal{T},X,Y) \wedge p(\mathcal{T},Y) \wedge \mathrm{succ}(\mathcal{T},\mathcal{T}').$	`p(X)@next :- q(X, Y), p(Y).`
Initial Facts: replacing the time argument 0, the suffix "@0" is added.	
$p(0,5).$	`p(5)@0.`
Static Facts: We leave the body empty. *time* is a reserved predicate defined by $time(0)$ and $time(\mathcal{T}') \leftarrow time(\mathcal{T}) \wedge \mathrm{succ}(\mathcal{T},\mathcal{T}')$. These two rules are added to every program. They ensure that *time* will be true for all states.	
$p(\mathcal{T},5) \leftarrow time(\mathcal{T}).$	`p(5).`
IO: We replace the call_-prefix, which can only appear in rule heads, with #. As the ret_-prefix can only appear in rule bodies, we replace that with # as well.	
$\mathrm{call\_f}(\mathcal{T}',X,?) \leftarrow p(\mathcal{T},X) \wedge \mathrm{succ}(\mathcal{T},\mathcal{T}').$	`#f(X, ?)@next :- p(X).`
$p(\mathcal{T},X) \leftarrow \mathrm{ret\_f}(\mathcal{T},5,X).$	`p(X) :- #f(5, X).`

We obtain a state for our system using the deductive rules. Through inductive rules some data, be it from existing facts or calls to external functions, is fed back into the system as the initial input for the next state deduction. This leads to a co-inductive model that allows for non-termination. And indeed we do not even allow for termination, as a microcontroller usually runs until it is turned off. We can interpret a Microlog program as a stream of function calls with their return values fed back into the system.

Of course, another interpretation is that the environment is the extensional database of a Datalog program and the call-convention just restricts which queries on this extensional database are allowed.

Definition 1 (Program Semantics)**.** The semantics of the Microlog program P_M is the mapping from input facts $\mathcal{E}$, a set of ret_-facts, to the minimal model $\mathcal{I}_{\text{min}}$ of $P := P_M \cup \mathcal{E}$, i.e., the set of all derivable facts.

We are actually not interested in arbitrary sets $\mathcal{E}$, but only sets satisfying the causality requirement that the ret_-facts in $\mathcal{E}$ correspond to derived call_-facts: Informally, a set of input facts is causal if every return value has a corresponding call that caused it, and every call has a corresponding return value with the results. There exists a bijective mapping between call_-facts and ret_-facts in the minimal model.

Even though Microlog can simulate Turing machines, and is therefore by itself computationally complete and could be used to implement any program or decision procedure, computational completeness on its own does not make a programming language generally useful to describe any program, as evidenced by humanity still inventing (or discovering) new abstractions and paradigms, and programming languages based upon them.

3 Microlog in the Agent Model

We want to classify the programs that can be written in Microlog. This gives us an idea of the power and expressivity of our language. We will compare Microlog against the classification scheme for agents put forward by Norvig and Russel. The definitions are lifted from [RN10] in an abbreviated form.

Definition 2 (Agent)**.** An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.

Definition 3 (Percept)**.** A percept is the agent's perceptual input at any given moment. In Microlog this is the subset of the input facts $\mathcal{E}$ having the same timestamp corresponding to the given point in time. We subscript $\mathcal{E}$ to select for the subset with a certain timestamp:

$$\mathcal{E}_i = \{\text{ret\_f}(c_1, \ldots, c_n) \mid \text{ret\_f}(i, c_1, \ldots, c_n) \in \mathcal{E}\}$$

Definition 4 (Percept Sequence)**.** An agent's percept sequence is the complete history of everything the agent has ever perceived. In Microlog this is a sequence of subsets of input facts:

$$\langle \mathcal{E}_0, \mathcal{E}_1, \ldots, \rangle$$

$\mathcal{E}_0$ is always empty for a causal set of input facts.

Definition 5 (Behaviour)**.** An agent's behaviour are the actions performed after a given sequence of percepts. In Microlog, this is a sequence of call facts:

$$\langle \mathcal{C}_0, \mathcal{C}_1, \ldots, \rangle$$

This is everything that is observable about an agent. Specifically we cannot examine the internal state of the agent. This also means that two agents, independent of their internal state or program, are indiscernible iff they behave the same for every sequence of percepts.

Definition 6 (Agent Program)**.** The mapping function from percepts (or the historical sequence thereof) to actions is called the agent program.
This mirrors definition 1, ignoring the unobservable part of a state.

Norvig and Russel outline five basic kinds of agent programs where we analyse whether we can define such a program in Microlog.

3.1 Simple Reflex Agents

Definition 7 (Simple Reflex Agent)**.** Simple reflex agents select actions on the basis of the current percept, ignoring the rest of the percept history.

The Microlog programs of this class simply map their inputs to outputs. The "Relay" program that just copies (or relays) some input to some output, for example a pressed button to a lit LED, is maybe the simplest possible program that uses both a sensor and an actuator.

Program 1. "Relay" program

```
(1)  #digitalRead(2, ?)@next.
(2)  #digitalWrite(13, State)@next :- #digitalRead(2, State).
```

And indeed, many microcontroller programs are simple reflex agents. Other examples from introductory robotics courses, where the function that maps inputs to outputs is more complex than that, are some of the Braitenberg vehicles [Ste85], or an edge-follower.

3.2 Model-Based Reflex Agents

Definition 8 (Model-Based Reflex Agent)**.** A model-based reflex agent maintains some sort of internal state that depends on the percept history and thereby reflects at least some of the unobserved aspects of the current state.

In the "Ventilation Control" program we control a greenhouse and want to close some ventilation system when temperatures go below 20 degrees and open the ventilation system when the temperatures go above 35 degrees, trying to hold the temperatures between those two points. The problem is, that the temperature sensor might give erroneous data and sometimes returns very high or very low temperatures that should not affect our normal operation.

In the program we read the temperature unconditionally in line `(1)` and deduce whether the sensor values are reasonable in lines `(2)` and `(3)`. If the read temperatures were reasonable, we commit them to the next state `(4)` and if they were not, we keep the old temperature, if there was one, in line `(5)`. The rest of the program is then committing to an action depending on the temperature that

was deemed reasonable. As an invariant, only temperatures deemed not invalid are only ever used as arguments to `temperature`-facts. Therefore the system is resilient against sensor errors. Whether a value is deemed valid or invalid could, of course, be augmented with more logic, expecting subsequent values to only differ by a certain amount, for example.

Program 2. "Ventilation Control" program

```
(1)  #readTemp(?)@next.
(2)  temperatureInvalid :- #readTemp(TNew), TNew < -10.
(3)  temperatureInvalid :- #readTemp(TNew), TNew > 100.
(4)  temperature(TNew)@next :- #readTemp(TNew), !temperatureInvalid.
(5)  temperature(TOld)@next :- temperature(TOld), temperatureInvalid.
(6)  #closeVentilation@next :- temperature(T), T < 20.
(7)  #openVentilation@next :- temperature(T), T > 35.
```

We can identify these kinds of agent programs by rules that conditionally commit sensor values to some set of predicates that form a world model. Only facts from those predicates are acted upon. On some conditions these facts are propagated into the future as they are identified as a kind of truth. Of course, this truth is updated as new facts and values that are consistent with previous knowledge are obtained.

Another example is the Toggle program that toggles a light whenever a button is pressed (pushed down and released). Here a part of the percept history is used to detect a high-low-edge on the button input (`isPressed` and `wasPressed`).

Program 3. "Toggle" program

```
(1)  #digitalRead(12, ?)@next.
(2)  isPressed :- #digitalRead(12, #HIGH).

(3)  wasPressed@next :- isPressed.
(4)  isReleased :- wasPressed, !isPressed.
(5)  lightOn@next :- isReleased, !lightOn.
(6)  lightOn@next :- lightOn, !isReleased.

(7)  #digitalWrite(13, #HIGH)@next :- lightOn.
(8)  #digitalWrite(13, #LOW)@next :- !lightOn.
```

3.3 Goal-Based Agents

Definition 9 (Goal-Based Agent)**.** A goal-based agent uses, besides the current state description, some sort of goal information that describes situations that are desirable. The agent program can combine this with the model (the same information as with the model-based reflex agent) to choose actions that achieve the goal.

Usually a goal is achieved by executing a plan. Planning, as in "creating a plan" is not always associated with Datalog. Intuitively a plan is a sequence of actions to take and as normal Datalog does not allow creation and manipulation

of lists, we can not easily model a plan this way. And for complex planning methods a different language might be suitable. We will still show an example how we can employ Microlog in a simple path planning application where the agent should find its way out of a maze.

In the program we will leave out the actions that read the current position or orientation in the maze. We assume that the agent "knows" its position either through given initial facts or suitable sensors. Movement between maze positions is achieved by deriving normal facts. Of course, for the actual movement of a robot, we would need to derive some actions to execute from a derived `go` fact.

We assume that initially we are in some position and we have a static maze and goal. Once the goal is reached we will deduce `finished` and we can assume that the agent plays a victorious fanfare through its speakers.

Our example maze (see Fig. 1) is a 4x4 grid labelled 1 to 16 with an `edge` relation that defines which ways are allowed to be moved through. We can imagine there to be walls, where no connection exists. We only define one direction and complete the symmetric edges through rule `(9)`. We have an initial `position` fact `(11)` and always deduce all possible directions that we can take `(14)`. We use the transitive closure to generate all possible destinations from there without backtracking to our current position `(16)`. Any path option that includes `goal` is `viable (17)`. In a maze without loops there is only ever one viable choice. This choice is taken `(19)` and in the next step the position is advanced `(20)` and the deduction starts anew until `goal` and `position` agree `(22)`.

Program 4. "Mazerunner" program

```
 (1)  % horizontal connections
 (2)  edge(1, 2).   edge(3, 4).   edge(6, 7).
 (3)  edge(7, 8).   edge(9, 10).  edge(10, 11).
 (4)  edge(11, 12). edge(13, 14). edge(14, 15).

 (5)  % vertical connections
 (6)  edge(1, 5).   edge(2, 6).   edge(3, 7).
 (7)  edge(8, 12).  edge(9, 13).  edge(12,16).

 (8)  % edge completion
 (9)  edge(X, Y) :- edge(Y, X).

(10)  % inital position and goal
(11)  position(2)@0.
(12)  goal(15).

(13)  % path finding
(14)  option(X) :- !finished, position(P), edge(P, X).
(15)  reachable(O, O) :- option(O).
(16)  reachable(O, X) :- reachable(O, P), edge(P, X), !position(X).
(17)  viable(O) :- option(O), goal(G), reachable(O, G).

(18)  % execution and finished
(19)  go(O) :- viable(O).
(20)  position(New)@next :- position(Old), go(New), edge(Old, New).
(21)  position(Old)@next :- position(Old), !go(_).
(22)  finished :- goal(G), position(G).
```

We can see that even though we can not formulate a full plan with all the positions that need to be reached in order, we can deduce the prefix of a plan by identifying viable directions from which the goal is reachable and take that direction.

If there were multiple choices (for example, if the maze had a loop on which the agent is on), we could decide on one by taking the lowest or highest valued one (by coordinate). The problem is, that this might throw us into a movement loop. We can add rules to save all positions where we have been and never search into or through those positions again.

This method also allows us to dynamically create an obstacle map. Starting from a fully connected grid, we can "hide" connections if, for example, a distance sensor sees a wall in front of it (say we deduce an `obstructed`-fact from some IO). With a new obstacle, the agent needs to forget all `beenThere`-facts though, as it might need to backtrack, since it could have unknowingly moved into a dead end. Through line `(17)` the position is retained and through line `(19)` the old `beenThere`-facts are discarded, in order to allow backtracking to circumnavigate the newly observed obstacle.

Program 5. "Obstacle-Finding Mazerunner" program

```
(1)  % obstacle finding
(2)  newObstacle(P, O) :- position(P), edge(P, O),
     obstructed(O), !knownObstacle(P, O).
(3)  knownObstacle(X, Y)@next :- knownObstacle(X, Y).
(4)  knownObstacle(X, Y)@next :- newObstacle(X, Y).
(5)  freeEdge(X, Y) :- edge(X, Y),
     !newObstacle(X, Y), !knownObstacle(X, Y).

(6)  % path finding (only using free edges, as far as we know)
(7)  option(X) :- !finished, position(P), freeEdge(P, X), !beenThere(X).
(8)  reachable(O, O) :- option(O).
(9)  reachable(O, X) :- reachable(O, P), freeEdge(P, X), !beenThere(X).
(10) viable(O) :- option(O), goal(G), reachable(O, G).

(11) % option selection (same as before)
(12) lesserViable(P1) :- viable(P1), viable(P2), P1 < P2.
(13) oneViable(P1) :- viable(P1), !lesserViable(P1).
(14) go(O) :- oneViable(O).

(15) % execution, memoization and finished
(16) position(New)@next :- position(Old), go(New), edge(Old, New).
(17) position(Old)@next :- position(Old), !go(_).
(18) beenThere(P)@next :- position(P).
(19) beenThere(P)@next :- beenThere(P), !newObstacle(_, _).
(20) finished :- goal(G), position(G).
```

From the example we can see that we can indeed write goal-based agents in Microlog. We can reason about the reachable states if an action were to be taken and identify viable actions. Such programs can be identified by rules that collect possible actions, and simulate actions taken. In this example, the `option` relation

contains all actions our agent can take and the `reachable` relation contains all possible `position` consequences after that specific action is taken.

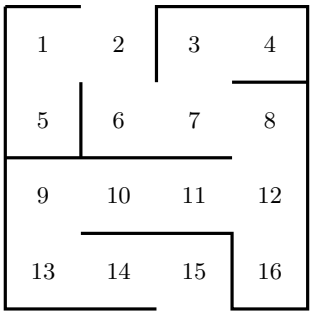

Fig. 1. 4 by 4 Maze

3.4 Utility-Based Agents

Definition 10 (Utility-Based Agent)**.** A utility-based agent includes a general performance measure that allow a comparison of different plans according to exactly how happy they would make the agent, because they are quicker, safer, more reliable, or cheaper than others.

With this agent description we can see that we reach difficulties with the expressivity that is allowed in Microlog. As we have seen from the previous example, it is not a problem if there are different plans, as we can identify prefixes of viable plans and execute this prefix, making other plans no longer viable through memoization of previous positions or states. What we do not have is a representation of one or more full plan. This makes it impossible to weigh the plans against each other. Specifically we cannot distinguish between two different plans with the same prefix, even if we had some way to measure and compare the quality of different plans.

So according to the classification by Norvig and Russel, Microlog is capable as an implementation language for agent programs for goal-based agents (and less powerful agents) but not utility-based agents. This is due to the fact that we have no representation of full plans and therefore no utility function to measure plans against each other. We will not discuss "learning agents", the most powerful agent class defined.

Through external function calls any operation, decision procedure, or algorithm may be attached, using a sufficiently equipped external software or hardware frameworks and mechanisms. But when investigating the expressivity of Microlog itself, taking a computationally complete host language (and arbitrarily powerful oracle machines that might be attached to the agent) into account as well, can reasonably be considered "cheating".

4 Representation of State and Transitions

Microlog has a notion of state that is used to encode the state of an agent program. In Microlog a state is a set of facts deduced for a specific timestamp. According to the inductive rules some data and facts, as well as call-directives, are transported into the next state and are the basis for further deductions.

One representation of the states is a generic approach, where facts are stored in an organized fashion within a database system. As the the number of relations and the number of their arguments are known, various database-approaches are possible. Mozilla's now archived Datalog engine "Mentat"[1] uses SQLite as its database backend. The Datalog-based language Soufflé uses special memory-saving data structures for faster parallel access and storage of special relation types, like equivalence relations. [Nap+19] Other generic backends for fact storage may be used. For Microlog we describe this compilation approach in [WB19]. Our compiler, which is optimized to generate as little data structure overhead (i. e., pointers) as possible, uses a single continuous buffer to store facts of a state (see Figure 2).

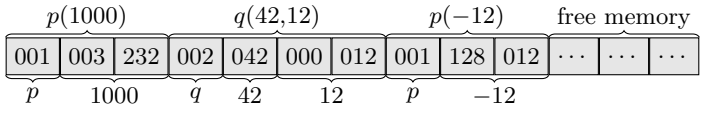

Fig. 2. Mapping Example with Declarations `.decl p(int)`, `.decl q(byte, int)`

Russel and Norvig describe this as the "structured representation", where a state includes objects, each of which may have attributes of its own as well as relationships to other objects. [RN10]. We use this compilation technique when our compiler can not deduce that the Microlog program uses for all possible environments just a finite amount of memory. Of course, as Microlog is computationally complete, this is undecidable in general. On the other hand, if all external functions can be simulated with Datalog rules (i. e., the output arguments are constants or projections of the input arguments), the memory needed is always finite.

If our compiler deduces that there is only a finite amount of facts for each reachable state, a different compilation technique is used. With a finite amount of facts per state, these states can be enumerated. States with the same number and kind of facts may still differ in some value obtained from an external function call.

For the Ventilation Control program, for example, both {`temperature(25), #readTemp(?)`} and {`temperature(30), #readTemp(?)`} are reachable states (up to timestamp). As this value has previously been obtained through a `#readTemp`-call and been written to a specific memory location, both states may refer to the

[1] `https://github.com/mozilla/mentat`

same location of the return value (`V1`) and collapse to the state {`temperature(V1)`, `#readTemp(?)`}. The result is a finite state machine that uses additional memory locations to store values obtained from external function calls, in order to combine sets of states that only differ in concrete return values. The transition function of the finite state machine may refer to these memory locations. We describe this compilation technique in [WB20].

An example state machine for the Toggle program is shown in Figure 3. All states besides the initial state contain the result of the `#digitalRead`-call in slot V_1. Upon entering states II to IV the LED is turned off, upon entering states V to VII it is turned on. The transition depends on the value obtained from the `#digitalRead`-call, with the dashed line meaning V_1 being `#LOW` and for the solid line it being `#HIGH`. The number of states we obtain is within the expected range, as we need two bits to detect the high-low-edge of the button press, in order to detect its release, and another bit to store whether the light should switch from on to off, or from off to on.

Russel and Norvig describe this as the "factored representation", where a state consists of a vector of attribute values. In our case we use global memory locations to store these attribute values, and a state refers to a subset of those. If a value is transported into the next state then no copying of that value is necessary, as the subsequent state just refers to the same global memory location.

Russel and Norvig also describe the even simpler "atomic representation", where a state is just a black box with no internal structure. Though we do not have a compiler explicitly for this representation, the finite state machine compiler does exactly that, if no external value is ever obtained. This is possible for programs where the external functions called offer no return value, i. e., the agent never senses its environment and only acts in a pre-described sequence.

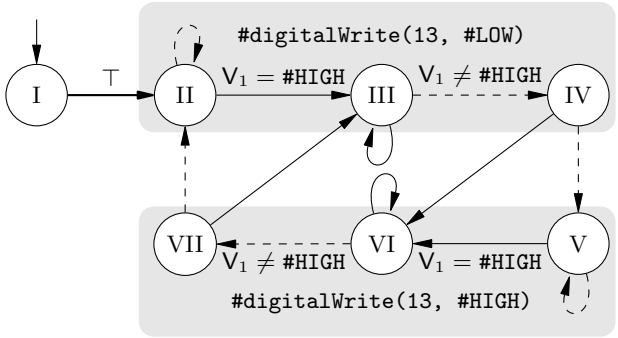

Fig. 3. Visualisation for Abstract Execution of the Toggle Program

5 Conclusion

We have shown that Microlog may be used to encode three of the five types of intelligent agents as defined by Norvig and Russel in an intuitive manner. Even though Microlog is computationally complete, the lack of native support for growing data structures, such as lists, and functions to deconstruct them, such as a fold, precludes an intuitive encoding of planned actions as sequences, and cost or value as functions thereof.

The agent types we presented as feasible were the simple reflex agent, the model-based reflex agent, and the goal-based agent. The simple reflex agent just maps its inputs to its outputs, like the Relay program. The model-based reflex agent, such as the Ventilation Control, is robust against erroneous sensor readings or a physical limitation on the amount of data that can be collected from the environment at a time. Through an internal world-model that is updated as new knowledge is obtained, the output is a function of that model. The goal-based agent, like the Mazerunner, besides its internal state, has a representation of its goal (in this case quite literal the `goal` relation). The output of the agent is a function of both the internal state and the goal that is to be reached or achieved.

We have shown that, for the goal-based agent, we can not model and store an arbitrarily sized plan that obtains the goal, but we can check finitely sized plan-prefixes whether plans starting with exactly that prefix would lead to the desired situation. Such a prefix is then executed until the goal is reached.

As we only ever obtain plan-prefixes and the information, whether this prefix leads to the goal or not, we can not weigh the costs of plans against each other. This prevents us from encoding a utility-based agent, since we have no way to apply the utility-function on a plan. Of course, through a lot more code it must be possible to do it, as Microlog is computationally complete. But the utility of a programming language stems from it allowing the implementation of decision procedures in an intuitive manner. We have shown that Microlog is expressive and its use-cases go beyond toy programs. Its deductive capabilities and memory-managed nature are certainly useful in a space where imperative "on-the-metal" C++-programming is the norm. Even broader we have shown that this model of side-effects and the strict call-convention we propose is useful as well.

Our compiler, that supports both compilation techniques and additional program transformations not shown here, as well as further example programs [2] for Arduino and the LEGO EV3 robotics platform, further demonstrating the viability of our approach, is available at `https://dbs.informatik.uni-halle.de/microlog/`.

[2] Like the Turing machine template, or the edge-follower

References

[Alv+10] Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. "Dedalus: Datalog in Time and Space". In: *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*. Ed. by Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers. Vol. 6702. Lecture Notes in Computer Science. Springer, 2010, pp. 262–281. DOI: 10.1007/978-3-642-24206-9_16. URL: https://doi.org/10.1007/978-3-642-24206-9%5C_16.

[Bas+10] Selen Basol, Ozan Erdem, Michael Fink, and Giovambattista Ianni. "HEX Programs with Action Atoms". In: *Technical Communications of the 26th International Conference on Logic Programming, ICLP 2010, July 16-19, 2010, Edinburgh, Scotland, UK*. Ed. by Manuel V. Hermenegildo and Torsten Schaub. Vol. 7. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2010, pp. 24–33. DOI: 10.4230/LIPIcs.ICLP.2010.24. URL: https://doi.org/10.4230/LIPIcs.ICLP.2010.24.

[CI05] Francesco Calimeri and Giovambattista Ianni. "External Sources of Computation for Answer Set Solvers". In: *Logic Programming and Nonmonotonic Reasoning, 8th International Conference, LPNMR 2005, Diamante, Italy, September 5-8, 2005, Proceedings*. Ed. by Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina. Vol. 3662. Lecture Notes in Computer Science. Springer, 2005, pp. 105–118. DOI: 10.1007/11546207_9. URL: https://doi.org/10.1007/11546207%5C_9.

[Enn82] Richard Ennals. "Teaching Logic as a Computer Language in Schools". In: *Proceedings of the First International Logic Programming Conference, Faculté des Science de Luminy, ADDP-GIA, Marseille, France, September, 14-17, 1982*. Ed. by Michel Van Caneghem. ADDP-GIA, 1982, pp. 99–104.

[HY90] Richard Hull and Masatoshi Yoshikawa. "ILOG: Declarative Creation and Manipulation of Object Identifiers". In: *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*. Ed. by Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek. Morgan Kaufmann, 1990, pp. 455–468. URL: http://www.vldb.org/conf/1990/P455.PDF.

[Nap+19] Patrick Nappa, David Zhao, Pavle Subotic, and Bernhard Scholz. "Fast Parallel Equivalence Relations in a Datalog Compiler". In: *28th International Conference on Parallel Architectures and Compilation Techniques, PACT 2019, Seattle, WA, USA, September 23-26, 2019*. IEEE, 2019, pp. 82–96. DOI: 10.1109/PACT.2019.00015. URL: https://doi.org/10.1109/PACT.2019.00015.

[RN10] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach, Third International Edition*. Pearson Education, 2010.

ISBN: 978-0-13-207148-2. URL: http://vig.pearsoned.com/store/product/1,1207,store-12521%5C_isbn-0136042597,00.html.

[SW19] Sibylle Schwarz and Mario Wenzel. "ev3dev-prolog - Prolog API for LEGO EV3". In: *49. Jahrestagung der Gesellschaft für Informatik, 50 Jahre Gesellschaft für Informatik - Informatik für Gesellschaft, INFORMATIK 2019 - Workshops, Kassel, Germany, September 23-26, 2019*. Ed. by Claude Draude, Martin Lange, and Bernhard Sick. Vol. P-295. LNI. GI, 2019, pp. 385–398. DOI: 10.18420/inf2019_ws41. URL: https://doi.org/10.18420/inf2019%5C_ws41.

[Ste85] Mark Stefik. "V. Braitenberg, Vehicles: Experiments in Synthetic Psychology". In: *Artif. Intell.* 27.2 (1985), pp. 246–248. DOI: 10.1016/0004-3702(85)90057-8. URL: https://doi.org/10.1016/0004-3702(85)90057-8.

[WB19] Mario Wenzel and Stefan Brass. "Declarative Programming for Microcontrollers - Datalog on Arduino". In: *Declarative Programming and Knowledge Management - Conference on Declarative Programming, DECLARE 2019, Unifying INAP, WLP, and WFLP, Cottbus, Germany, September 9-12, 2019, Revised Selected Papers*. Ed. by Petra Hofstedt, Salvador Abreu, Ulrich John, Herbert Kuchen, and Dietmar Seipel. Vol. 12057. Lecture Notes in Computer Science. Springer, 2019, pp. 119–138. DOI: 10.1007/978-3-030-46714-2_9. URL: https://doi.org/10.1007/978-3-030-46714-2%5C_9.

[WB20] Mario Wenzel and Stefan Brass. "Translation of Interactive Datalog Programs for Microcontrollers to Finite State Machines". In: *Logic-Based Program Synthesis and Transformation - 30th International Symposium, LOPSTR 2020, Bologna, Italy, September 7-9, 2020, Proceedings*. Ed. by Maribel Fernández. Vol. 12561. Lecture Notes in Computer Science. Springer, 2020, pp. 210–227. DOI: 10.1007/978-3-030-68446-4_11. URL: https://doi.org/10.1007/978-3-030-68446-4%5C_11.